Bounded Auditable Restoration of Distributed Systems*

Reza Hajisheykhi, Mohammad Roohitavaf, Sandeep S. Kulkarni
Computer Science and Engineering Department
Michigan State University
East Lansing, MI, USA
Email: {hajishey, roohitav, sandeep}@cse.msu.edu

Abstract

We focus on protocols for auditable restoration of distributed systems. The need for such protocols arises due to conflicting requirements (e.g., access to the system should be restricted but emergency access should be provided). One can design such systems with a tamper detection approach (based on the intuition of *In-case-of-emergency-break-glass*). However, in a distributed system, such tampering, which are denoted as auditable events, is visible only for a single node. This is unacceptable since the actions they take in these situations can be different than those in the normal mode. Moreover, eventually, the auditable event needs to be cleared so that system resumes the normal operation.

With this motivation, in this paper, we present two protocols for auditable restoration, where any process can potentially identify an auditable event. The first protocol has an unbounded state space while the second protocol uses bounded state space that does not increase with the length of the computation. In both protocols, whenever a new auditable event occurs, the system must reach an *auditable state* where every process is aware of the auditable event. Only after the system reaches an auditable state, it can begin the operation of restoration. Although any process can observe an auditable event, we require that only *authorized* processes can begin the task of restoration. Moreover, these processes can begin the restoration only when the system is in an auditable state. Our protocols are self-stabilizing and can effectively handle the case where faults or auditable events occur during the restoration protocol. Moreover, they can be used to provide auditable restoration to other distributed protocols.

I. Introduction

Stabilization is a type of fault-tolerance that requires that the system must recover from any transient fault. By contrast, tamper evidence requires that any tampering cannot be erased. Hence, these requirements are by definition contradictory. The former requires that the effect of all faults be erased whereas the latter requires it to be permanent. Our goal in this paper is to combine these two aspects of dependability especially to support events –denoted by auditable events– that are caused by conflicting requirements. We present two protocols that provide stabilization and handle auditable events. In both protocols, if some faults (e.g., transient faults, process failure) occur, the system restores to its legitimate states. However, if the system is perturbed by auditable events then the system reaches an *auditable* state, i.e., a state where the program provides potential different behavior (e.g., reduced functionality) and where all processes are permanently aware of that auditable event. Starting from an auditable state, an authorized process can restore the system to its original state. Moreover, the protocols are stabilizing in that even if they variables are corrupted by transient faults, they recover themselves to a state from where subsequent auditable events are handled correctly.

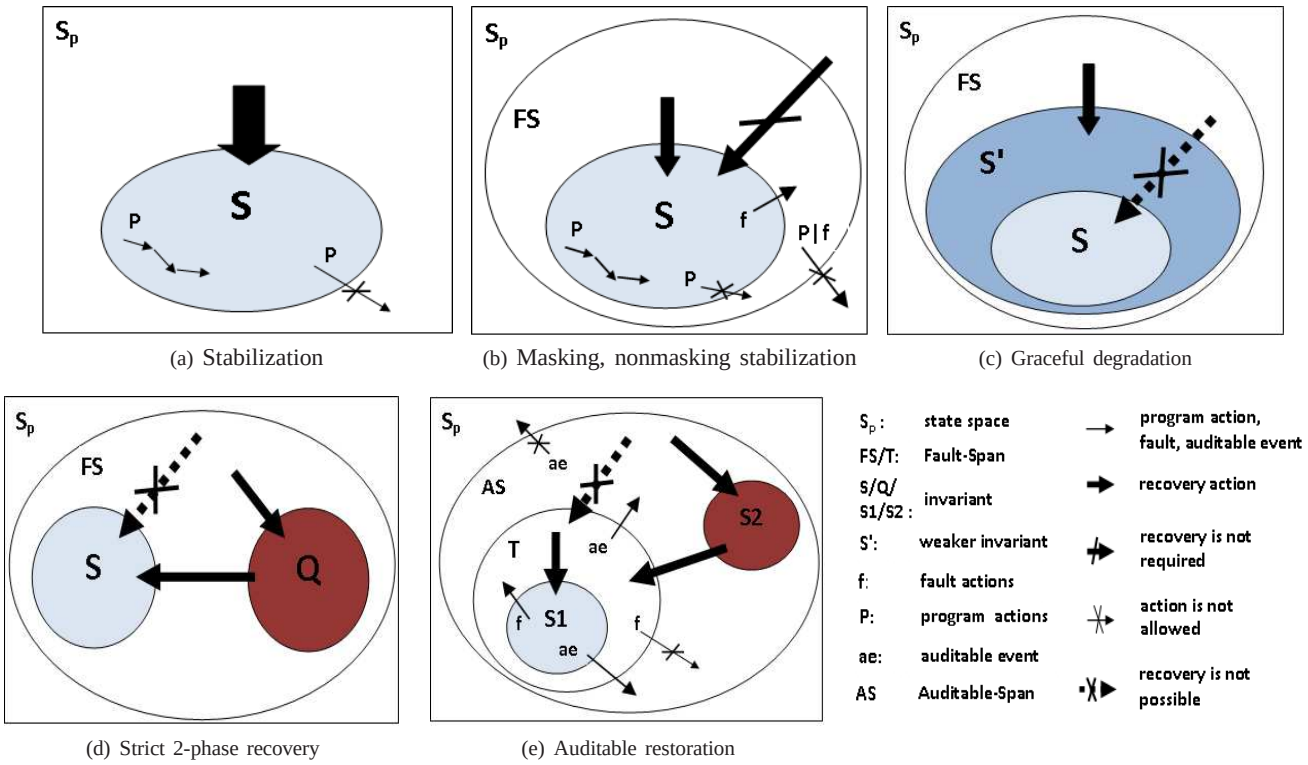
A. A Brief History and the Need for Auditable Restoration

Fault-tolerance focuses on the problem of what happens if the program is perturbed by undesired perturbations. In other words, it focuses on what happens if the program is perturbed beyond its legitimate states (a.k.a. *invariant*). There have been substantial ad-hoc *operational* approaches –designed for specific types of faults– for providing fault-tolerance. For example, the idea of recovery blocks [**?**] introduced the notion of acceptance conditions that should be satisfied at certain points in the computation. If these conditions are not satisfied, the program is restored to the previous state from where another recovery block is executed. Checkpointing and recovery based approaches provide mechanism to restore the program to a previous checkpoint. Dijkstra [**?**] introduced an approach for *specifying* (i.e., identifying what the program should provide irrespective of how it is achieved) one-type of fault-tolerance, namely stabilization [**?**]. A stabilizing program partitioned the state space of the program into legitimate states (predicate S in Figure **??**) and other states. It is required that (1) starting from any state in S, the program always stays in S and (2) starting from any state in $\neg S$, the program recovers to a state in S.

Although stabilization is desirable for many programs, it is not suitable for some programs. For example, it may be impossible to provide recovery from all possible states in $\neg S$. Also, it may be desirable to satisfy certain safety properties during recovery. Arora and Gouda [**?**] introduced another approach for formalizing a fault-tolerant system that ensures *convergence* in the presence of transient faults (e.g., soft errors, loss of coordination, bad initialization), say f. That is, from any state/configuration, the system recovers to its invariant S, in a finite number of steps. Moreover, from its invariant, the executions of the system satisfy its specifications and remain in the invariant; i.e., *closure*. They distinguish two types of fault-tolerant systems: *masking* and *nonmasking*. In the former, the effects of failure is completely invisible to the application. In other words, the invariant is equal to the fault-span ($S = FS$ in Figure **??**). In the latter, the fault-affected system may violate the invariant but the continued execution of the system yields a state where the invariant is satisfied (See Figure **??**).

The approach by Arora and Gouda intuitively requires that after faults stop occurring, the program provides the *original functionality*. However, in some cases, restoring the program to *original* legitimate states so that it satisfies the subsequent specification may be impossible. Such a concept has been considered in [**?**], where authors introduce the notion of *graceful degradation*. In graceful degradation (cf. Figure **??**), a system satisfies its original specification when no faults have occurred. After occurrence of faults (and when faults stop

*A preliminary version of this work appears in SRDS'15 conference.



(a) Stabilization (b) Masking, nonmasking stabilization (c) Graceful degradation

(d) Strict 2-phase recovery (e) Auditable restoration

Fig. 1: Different types of stabilizations.

occurring), it may not restore itself to the original legitimate states (S in Figure **??**) but rather to a larger set of states (S' in Figure **??**) from where it satisfies a weaker specification. In other words, in this case, the system may not satisfy the original specification even after faults have stopped.

In some instances, especially where the perturbations are *security related*, it is not sufficient to restore the program to its original (or somewhat degraded) behavior. The notion of multi-phase recovery was introduced for such programs [**?**]. Specifically, in these programs, it is necessary that recovery is accomplished in a sequence of phases, each ensuring that the program satisfies certain properties. One of the properties of interest is *strict 2-phase recovery*, where the program first recovers to the set of states Q that are strictly disjoint from legitimate states. Subsequently, it recovers to legitimate states (See Figure **??**).

The goal of auditable restoration is motivated by combining the principles of the strict 2-phase recovery and the principles of fault-tolerance. Intuitively, the goal of auditable restoration is to classify system perturbations into faults and auditable events, provide fault-tolerance (similar to that in Figure **??**) to the faults and ensure that strict 2-phase recovery is provided for auditable events. Unfortunately, this cannot be achieved for arbitrary auditable events since in [**?**], it has been shown that adding strict 2-phase recovery to even a centralized program is NP-complete. Our focus in this paper is on auditable events that are (immediately) detectable while faults may or may not be detectable.

B. Goals of Auditable Restoration

In our work, we consider the case that the system is perturbed by possible faults and possible tampering that we call as *auditable events*. Given that both of these are perturbations of the system, we distinguish between them based on how and why they occur. By faults, we mean events that are random in nature. These include process failure, message losses, transient faults, etc. By auditable events, we mean events that are deliberate in nature for which a detection mechanism has been created. Among other things, the need for managing such events arises due to conflicting nature of system requirements. For example, consider a requirement that states that each process in a distributed system is physically secure. This requirement may conflict with another requirement such as each node must be provided emergency access (e.g., for firefighters). As another example, consider the requirement that each system access must be authenticated. This may conflict with the requirement for (potential) unauthorized access in a crisis. Examples of this type are well-known in the domain of power systems, medical record systems, etc., where the problem is solved by techniques such as writing down the password in a physically secure location that can be broken into during crisis or by allowing unlimited access and using logs as a deterrent for unauthorized access. Yet another example includes services such as Gmail that (can) require that everytime a user logs in, he/she can authenticate via 2-factor authentication such as user's cell phone. To deal with situations where the user may not have access to a cell phone, the user is provided with a list of 'one-time' passwords and it is assumed that these will always stay in the control of the users. However, none of these solutions are fully satisfactory.

In this paper, we focus on a solution that is motivated by the notion of 2-phase recovery. Specifically, we would like to have the following properties:

- We consider the case where the auditable events are immediately detectable. This is the case in all scenarios discussed above. For example, the use of 'one-time' password for Gmail or violation of physical security of a

process is detectable. Likewise, the passwords stored in a physically secure location can be different from those used by ordinary users making them detectable.

- We require that if some process is affected by an auditable event, then eventually all (respectively, relevant)[1] processes in the system are aware of this auditable event. For example, if one process is physically tampered with then it will automatically cause the tampering to be detected at other processes. This would allow the possibility that they only provide the 'emergency' services and protect the more sensitive information. This detection must occur even in the presence of faults (except those that permanently fail all processes that were aware of auditable event)[2]. We denote such states as *auditable states* in that all processes are aware of a new auditable event ($S2$ in Figure **??**).
- After all processes are aware of the auditable event, there exists at least one process that can begin the task of restoring the system to a normal state, thereby clearing the auditable event. This operation may be automated or could involve human-in-the-loop. However, the system should ensure that this operation cannot be initiated until all processes are aware of the auditable event. This ensures that any time the normal operation is restored, all processes are aware of the auditable event.
- The operation to restore the system to normal state succeeds even if it is perturbed by faults such as failure of processes. However, if an auditable event occurs while the system is being restored to the normal state, the auditable event has a higher priority, i.e., the operation to restore to normal state would be canceled until it is initiated at a later time.

Utilizing such a solution, which is a variation of strict 2-phase recovery, we propose two protocols, AR and BAR. The first protocol, AR, has an unbounded state space while the second protocol, BAR, utilizes a bounded state space. The space maintained by BAR is $O(log\ \ N)$, where N is the number of processes in the system. In our protocols, when an auditable event occurs, the system is guaranteed to recover to a state where all processes are aware of this event ($S2$ in Figure **??**). And, subsequently, the system recovers to its normal legitimate states ($S1$ in Figure **??**). Our protocols have the following properties:

- They are self-stabilizing. If they are perturbed to an arbitrary state, they will recover to a state from where all future auditable events will be handled correctly.
- Our protocols ensure that if auditable events occur at multiple processes *concurrently*, it will be treated as one event restoring the system to the auditable state. However, if an auditable event occurs after the system restoration to normal states has begun (or after system restoration is complete), it will be treated as a new auditable event.
- After the occurrence of an auditable event, the system recovers to the *auditable state* even if it is perturbed by failure of processes, failure of channels, as well as certain transient faults.
- No process can initiate the restoration to normal operation unless the system is in an auditable state. In other words, in Figure **??**, a process cannot begin restoration to normal state unless the system was recently in $S2$.
- After the system restores to an auditable state and some process initiates the restoration to normal operation, it completes correctly even if it is perturbed by faults such as failure of processes or channels. However, if it is perturbed by an auditable event, the system recovers to the auditable state again.
- We show how AR can be modified to prevent perturbation of variables used in AR. We also show that BAR can be implemented with a finite state space and the total number of states does not increase with the length of the computation. It is well-known that achieving finite state space for self-stabilizing programs is difficult [**?**].

Organization of the paper. The rest of the paper is organized as follows: In Section **??**, we present the preliminary concepts of stabilization and fault-tolerance. The notion of auditable restoration is formally defined in Section **??**. In Section **??**, we explain the unbounded auditable restoration protocol, AR, and illustrate it with an example in Section **??**. In Section **??**, we prove that AR is correct. Section **??** discusses how to prevent perturbation of variables used in AR. In Section **??**, we present the bounded auditable restoration protocol, BAR, and its proofs such that all variables used in BAR are bounded. Section **??** explains the applications of the auditable restoration protocols. Section **??** provides related work, and, finally, we conclude in Section **??**. Additionally, we provide a discussion section in Appendix **??**.

II. Preliminaries

In this section, we first recall the formal definitions of programs, faults, auditable events, and stabilization adapted from [**?**], [**?**], [**?**]. We then formally define our proposed auditable restoration mechanism.

Definition 1 (Program): A program p is specified in terms of a set of variables V, each of which is associated with a domain of possible values, and a finite set of actions of the form $\langle name \rangle :: \langle guard \rangle \longrightarrow \langle statement \rangle$

where $guard$ is a Boolean expression over program variables, $statement$ updates the program variables.

For such a program, say p, we define the notion of state, state space and transitions next.

Definition 2 ((Program) State and State Space): A state of the program is obtained by assigning each program variable a value from its domain. The state space (denoted by S_p) of such program is the set of all possible states.

Definition 3 (Enabled): We say that an action $g \longrightarrow st$ is enabled in state s iff g evaluates to true in s.

Definition 4 (Transitions corresponding to an action): The transitions corresponding to an action ac of the form $ac :: g \longrightarrow st$ (denoted by δ_{ac}) is a subset of $S_p \times S_p$ and is obtained as $\{(s_0, s_1) | g$ is true in s_0, and s_1 is obtained executing st in state $s_0\}$.

Definition 5 (Transitions corresponding to a program): The transitions of program p (denoted by δ_p) consisting of actions $ac_1, ac_2, \cdots ac_m$ are

$\delta_p = \bigcup_{i=1}^{m} \delta_{ac_i} \cup \{(s,s) | ac_1, ac_2 \cdots ac_m$ are not enabled in $s\}$.

Remark 1: Observe that based on the above definition, for any state s, there exists at least one transition in δ_p that originates from s. This transition may be of the form (s, s).

For subsequent discussion, let the state space of program p be denoted by S_p and let its transitions be denoted by δ_p.

[1] In this paper, for simplicity, we assume that all processes are relevant.

[2] If all processes that are (directly or indirectly) aware of the auditable events fail then it is impossible to distinguish this from the scenario where these processes fail before the auditable events.

Definition 6 (Computation): We say that a sequence $\langle s_0, s_1, s_2, ...\rangle$ is a *computation* iff

- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p$

Definition 7 (Closure): A state predicate S is *closed* in p iff $\forall s_0, s_1 \in S_p :: (s_0 \in S \wedge (s_0, s_1) \in \delta_p) \Rightarrow (s_1 \in S)$.

Definition 8 (Invariant): A state predicate S is an *invariant* of p iff S is closed in p.

Remark 2: Normally, the definition of invariant (legitimate states) also includes a requirement that computations of p that start from an invariant state are correct with respect to its specification. The theory of auditable restoration is independent of the behaviors of the program inside legitimate states. Instead, it only focuses on the behavior of p outside its legitimate states. We have defined the invariant in terms of the closure property alone since it is the only relevant property in the definitions/theorems/examples in this paper.

Definition 9 (Convergence): Let S and T be state predicates of p. We say that T *converges* to S in p iff

- $S \subseteq T$,
- S is closed in p,
- T is closed in p, and
- For any computation σ (=$\langle s_0, s_1, s_2, ...\rangle$) of p if $s_0 \in T$ then there exists l such that $s_l \in S$.

Definition 10 (Stabilization): We say that program p is *stabilizing* for invariant S iff S_p converges to S in p.

Definition 11 (Faults): Faults for program $p = \langle S_p, \delta_p\rangle$ is a subset of $S_p \times S_p$; i.e., the faults can perturb the program to any arbitrary state.

Definition 12 (Auditable Events): Auditable events for program $p = \langle S_p, \delta_p\rangle$ is a subset of $S_p \times S_p$.

Remark 3: Both faults and auditable events are a subset of $S_p \times S_p$. i.e., they both are a set of transitions. However, as discussed earlier, the goal of faults is to model events that are random in nature for which recovery to legitimate states is desired. By contrast, auditable events are deliberate. Additionally, there is a mechanism to detect auditable events and, consequently, we want an auditable restoration technique when auditable events occur.

Remark 4: We assume that the system utilizes a perfect failure detector mechanism as described in [**?**] that is essentially important. Additionally, similar to [**?**], we assume that if a process fails, all its neighbors can detect the failure.

Definition 13 (F-Span): Let S be the invariant of program p. We say that a state predicate T is a *f-span* of p from S iff the following conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $f \cup \delta_p$.

III. Defining Auditable Restoration

In this section, we formally define the notion of auditable restoration. The intuition behind this is as follows: Let $S1$ denote the legitimate states of the program. Let T be a fault-span corresponding to the set of faults f. Auditable events perturb the program outside T. If this happens, we want to ensure that the system reaches a state in $S2$. Subsequently, we want to restore the system to a state in $S1$.

Definition 14 (Auditable Restoration): Let ae and f be auditable events and set of faults, respectively, for program p. We say that program p is an *auditable restoration* program with auditable events ae and faults f for *invariant* $S1$ and auditable state predicate $S2$ iff there exists T

- T converges to $S1$ in p,
- T is closed in $\delta_p \cup f$,
- For any sequence $\sigma(= \langle s_0, s_1, s_2, ...\rangle)$ s.t.

$$
\begin{array}{l}
s_0 \in T \ \wedge (s_0, s_1) \in ae \ \wedge s_1 \notin T \ \wedge \\
\forall l : (s_l, s_{l+1}) \in (\delta_p \cup \delta_f \cup ae) \ \wedge \\
\exists w : \forall r : r \geq w \Rightarrow (s_r, s_{r+1}) \in \delta_p \\
\Rightarrow \forall x : ((s_x, s_{x+1}) \in ae \Rightarrow \\
\quad (\exists m, n : ((n > m \geq x) \wedge (s_m \in S2) \wedge (s_n \in S1)))
\end{array}
$$

IV. Stabilizing Auditable Restoration Protocol with Unbounded State Space

In this section we explain our unbounded protocol, AR, for distributed programs. As mentioned earlier, the auditable restoration protocol consists of several processes. Each process is potentially capable of detecting an auditable event. In other words, we permit each process to be able to detect auditable events but we do not require that each process is able to detect auditable events. However, only a subset of processes are capable of initiating the restoration operation. This captures the intuition that *clearing of the auditable event* is restricted to *authorized* processes only and can possibly involve human-in-the-loop. For simplicity, we assume that there is a unique process for this responsibility and the failure of this process is handled with approaches such as leader election [**?**].

The auditable restoration protocol is required to provide the following functionalities: (1) in any arbitrary state where no auditable event exists, the system reaches a state from where subsequent restoration operation completes correctly, (2) in any arbitrary state where some process detects the auditable event, eventually all processes are aware of this event, (3) some process in the system can detect that all processes have been aware of the auditable event, and (4) the process that knows all processes are aware of the auditable event starts a new restoration operation.

The auditable restoration protocol works as follows: It utilizes a stabilizing silent tree rooted at the leader process, i.e., it reaches a fixpoint state after building the tree even though individual processes are not aware of reaching the fixpoint. Several tree construction algorithms (e.g., [**?**]) can be utilized for this approach. The auditable restoration protocol is superimposed on top of a protocol for tree reconstruction, i.e., it only reads the variables of the tree protocol but does not modify them. The only variables of interest from the tree protocol are $P.j$ (denoting the parent of j in the tree) and $l.j$ (identifying the id of the process that j believes to be the leader in the tree). Additionally, we assume that whenever a tree action is executed, it notifies the auditable restoration protocol so it can take the corresponding action. Since the tree protocol is silent, some tree reconstruction is being done due to faults.

In addition, the program maintains the variable $otsn.j$ and $ctsn.j$. Intuitively, they keep track of the number of auditable events that j has been aware of and the number of auditable events after which the system has been restored by the leader process, respectively. Additionally, each process maintains $st.j$ that identifies its state, $sn.j$ that is a sequence number and $res.j$ that is $\{0..1\}$. In summary, each process j maintains the following variables:

- $P.j$, which identifies the parent of process j;
- $l.j$, which denotes the id of the process that j believes to be the leader;
- $st.j$, which indicates the state of process j. This variable can have four different values: *restore*, *stable*, $\perp$ (read *bottom*), or $\top$ (read *top*). These values indicate that j is in the middle of restoration after an auditable event has occurred, j has completed its task associated with

restoration, j is in the middle of reaching to the auditable state, or j has completed its task associated with reaching to the auditable state, respectively.
- $sn.j$, which denotes a sequence number;
- $otsn.j$, as describe above;
- $ctsn.j$, as described above, and
- $res.j$, which has the domain $\{0..1\}$.

Observe that in the above program, the domain of $sn.j$, $otsn.j$, and $ctsn.j$ is currently unbounded. This is done for simplicity of the presentation. We can bound the domain of these variables without affecting the correctness of the program. We discuss this issue in Section **??**.

The protocol AR consists of 11 actions. The first action, $AR1$, is responsible for detecting the auditable event. As mentioned above, each process has some mechanism that detects the auditable events. If an auditable event is detected, the process j increments $otsn.j$ and propagates it through the system.

Actions $AR2$–$AR5$ are for notifying all processes in the system about the auditable events detected. Specifically, action $AR2$ propagates the changes of $otsn$ value. We assume that this action is a high priority action and executes concurrently with every other action. Hence, for simplicity, we do not show its addition to the rest of the actions. When the leader process is notified of the auditable event, in action $AR3$, it changes its state to $\perp$ and propagates $\perp$ towards its children. Also, in action $AR3$, the leader sets its res variable to 1 by using $min(res.j + 1, 1)$ function. (Instead of setting res value to 1 directly, we utilize this function because it would simplify the presentation of protocol BAR in Section **??** by allowing us to reuse some actions from AR.) Actions $AR2$ and $AR3$ also need to coordinate with an application that uses this reset protocol to achieve auditable restoration. In particular, these actions should be synchronized with the application so that the application restores itself to an auditable state (i.e., state in $S2$ in Figure **??**). Action $AR4$ propagates $\perp$ towards the leaves. In action $AR5$, when a leaf receives $\perp$, it changes its state to $\top$ and propagates it towards the leader. Consider that, action $AR5$ detects whether all processes have participated in the current $\perp$ wave. This detection is made possible by letting each process maintain the variable res that is true only if its neighbors have propagated that wave. In particular, if process j has completed a $\perp$ wave with res false, then the parent of j completes that wave with the res false. It follows that when the leader completes the wave with the res true, all processes have participated in that wave. This action also increments the $ctsn$ value. However, if the leader fails when its state is $\perp$ and some process with state $\top$ becomes the new leader, this assumption will be violated. Action $AR3$ guarantees that, even if the leader fails, the state of all processes will eventually become $\top$ and the leader process will be aware of that.

Action $AR6$ broadcasts the changes of the $ctsn$ value so that the other processes can also restore to their legitimate states. We assume that, similar to action $AR2$, action $AR6$ is also a high priority action and executes concurrently with the other actions.

When the leader ensures that all processes are aware of the auditable events, a human input can ask the leader to initiate a restoration wave to recover the system to its legitimate state. Therefore, in action $AR7$, the leader initiates a distributed restoration wave, marks its state as *restore*, and propagates the restoration wave to its children.

When a process j receives a restoration wave from its parent,

AR1 :: $\{j$ *detects an auditable event* $\}$
$\longrightarrow otsn.j := otsn.j + 1$

AR2 :: $otsn.j < otsn.k$
$\longrightarrow otsn.j := otsn.k$, **if** $P.j = j$ **then** $res.j := 0$;
{*reset local state of* j}

AR3 :: $P.j = j \wedge st.j \neq \perp \wedge\ otsn.j > ctsn.j$
$\longrightarrow st.j, sn.j, res.j := \perp, sn.j + 1, min(res.j + 1, 1)$;
{*reset local state of* j}

AR4 :: $st.(P.j) = \perp \wedge sn.j \neq sn.(P.j) \wedge l.j = l.(P.j)$
$\longrightarrow st.j, sn.j, res.j :=$
$\perp, sn.(P.j), min(res.j + 1, 1)$

AR5 :: $(\forall k : P.k = j : otsn.k = otsn.j \wedge st.k = \top)$
$\wedge(\forall k : k \in Nbr.j : sn.j = sn.k \wedge l.j = l.k) \wedge st.j = \perp$
$\longrightarrow st.j := \top,$
$res.j := min(res.k)\ where\ k \in Nbr.j \cup \{j\}$
if $(P.j = j \wedge res.j \neq 1)$ **then**
$st.j, sn.j, res.j :=$
$\perp, sn.j + 1, min(res.j + 1, 1)$
else if $(P.j = j \wedge res.j = 1)$ **then**
$ctsn.j := otsn.j$

AR6 :: $ctsn.j < ctsn.k \longrightarrow ctsn.j = ctsn.k$
AR7 :: $P.j = j \wedge\ st.j = \top \wedge ctsn.j = otsn.j \wedge$
{*authorized to restore*}
$\longrightarrow st.j, sn.j := restore, sn.j + 1$;
{*reset local state of* j}

AR8 :: $st.(P.j) = restore \wedge sn.j \neq sn.(P.j) \wedge$
$l.j = l.k \wedge otsn.j = ctsn.j$
$\longrightarrow st.j, sn.j, res.j :=$
$restore, sn.(P.j), min(res.j + 1, 1)$;
{*reset local state of* j}

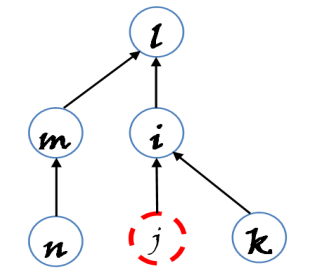
AR9 :: $(\forall k : P.k = j : sn.j = sn.k\ \wedge st.k = stable)\ \wedge$
$(\forall k : k \in Nbr.j : sn.j = sn.k \wedge l.j = l.k) \wedge$
$st.j = restore$
$\longrightarrow st.j := stable,$
$res.j := min(res.k) : k \in Nbr.j \cup \{j\}$
if $(P.j = j \wedge res.j \neq 1)$ **then**
$st.j, sn.j, res.j :=$
$restore, sn.j + 1, min(res.j + 1, 1)$
else if $(P.j = j \wedge res.j = 1)$ **then**
{*restore is complete*}

AR10 :: $\neg lc.j \longrightarrow st.j, sn.j := st.(P.j), sn.(P.j)$
AR11 :: $\langle$ *any tree correction action that affects process* j $\rangle$
$\longrightarrow res.j := 0$

Fig. 2: The unbounded auditable restoration protocol, AR.

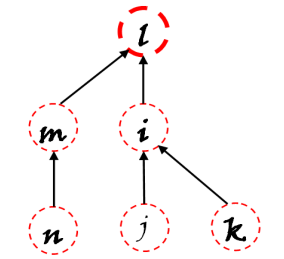
in action $AR8$, j marks its state as *restore* and propagates the wave to its children. Actions $AR7$ and $AR8$ also need to coordinate with the application so that the application state is restored to a stable state (i.e., state in $S1$ in Figure **??**).

When a leaf process j receives a restoration wave, j restores its state, marks its state as *stable*, and responds to its parent. In action $AR9$, when the leader receives the response from its children, the restoration wave is complete. Action $AR9$, like action $AR5$, detects if all processes participated in the current



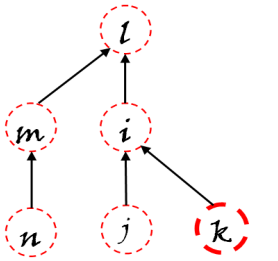
	sn	st	otsn	ctsn	res
l	0	stable	0	0	0
i	0	stable	0	0	0
j	0	stable	**1**	0	0
k	0	stable	0	0	0
m	0	stable	0	0	0
n	0	stable	0	0	0

(a) Node j detects an auditable event



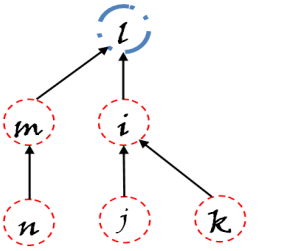
	sn	st	otsn	ctsn	res
l	1	⊥	1	0	1
i	0	stable	1	0	0
j	0	stable	1	0	0
k	0	stable	1	0	0
m	0	stable	1	0	0
n	0	stable	1	0	0

(b) The leader l initiates a $\perp$ wave



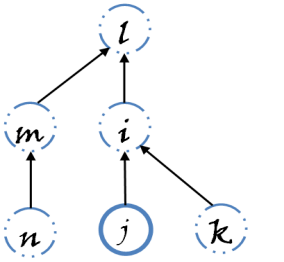
	sn	st	otsn	ctsn	res
l	1	⊥	1	0	1
i	1	⊥	1	0	1
j	1	⊥	1	0	1
k	1	⊤	1	0	1
m	1	⊥	1	0	1
n	1	⊥	1	0	1

(c) The leaf k receives the $\perp$ wave, changes its state to $\top$, and propagates it towards the leader



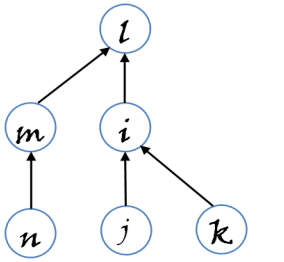
	sn	st	otsn	ctsn	res
l	2	**restore**	1	1	1
i	1	⊤	1	0	1
j	1	⊤	1	0	1
k	1	⊤	1	0	1
m	1	⊤	1	0	1
n	1	⊤	1	0	1

(d) The leader l initiates a $restoration$ wave



	sn	st	otsn	ctsn	res
l	2	restore	1	1	1
i	2	restore	1	1	1
j	2	**stable**	1	1	1
k	2	restore	1	1	1
m	2	restore	1	1	1
n	2	restore	1	1	1

(e) Node j propagates a $stable$ wave towards the leader



	sn	st	otsn	ctsn	res
l	2	stable	1	1	1
i	2	stable	1	1	1
j	2	stable	1	1	1
k	2	stable	1	1	1
m	2	stable	1	1	1
n	2	stable	1	1	1

(f) All nodes have changed their states to $stable$

Fig. 3: A step by step example illustrating the auditable restoration protocol AR.

restoration wave by utilizing the variable res. To represent action $AR10$, first, we define $lc.j$ as follows:

lc.j =

$$((st.(P.j) = restore \wedge st.j = restore) \Rightarrow sn.j := sn.(P.j) \wedge$$
$$st.(P.j) = stable \Rightarrow (st.j = stable \wedge sn.j = sn.(P.j)) \wedge$$
$$(st.(P.j) = \perp \wedge st.j = \perp) \Rightarrow sn.j = sn.(P.j) \wedge$$
$$st.(P.j) = \top \Rightarrow (st.j = \top \wedge sn.j = sn.(P.j)))$$

Action $AR10$ guarantees the stabilization of the protocol by ensuring that no matter what the initials state is, the program can recover to legitimate states from where future restoration operations work correctly. Finally, if any tree construction algorithm is called to reconfigure the tree and affects process j, action $AR11$ resets the res variable of process j.

V. An Illustrative Example

In this section, we present an example to better understand the protocol AR presented in Figure **??**. In this example, we assume that the system consists of 6 processes l, i, j, k, m, and n. Also, we assume that process l is selected as the leader of the system.

Let the initial state be as shown in Figure **??**. For simplicity, we assume that sn, $otsn$, $ctsn$, and res values are zero. Also, process j has detected an auditable event. Hence, it increments its $otsn$ value by executing action $AR1$. The computation proceeds as follows:

- The other nodes get notified of the auditable event and increment their $otsn$ values by executing action $AR2$. The leader l also gets notified of the auditable event by executing action $AR2$. Therefore, it initiates a $\perp$ wave by executing action $AR3$ and increments its $otsn$ and sn values (See Figure **??**).
- The $\perp$ wave propagates through the system towards the leaves. When a leaf, e.g. process k, receives it, the leaf first executes action $AR4$ and changes its state to $\perp$. Then, it executes action $AR5$ and changes its state to $\top$ and propagates the $\top$ wave towards the leader l (See Figure **??**).
- When the leader l receives the $\top$ wave, all processes in the system are aware of the auditable event. Moreover, since $res.l$ is 1, none of the processes in the system has failed. At this point, the leader l ensures that the system is in $S2$. Therefore, it initiates a restoration wave by executing action $AR7$ (See Figure **??**). Also action $AR6$ executes in concurrent with the other actions since the leader l has updated $ctsn.l$ value by $otsn.l$ value.
- The restoration wave goes through the system towards the leaves. When the leaf j receives this wave, first, it executes action $AR8$ and changes its state to restore. At this point, the system has been recovered to T (See

Figure **??**). Then, since all the neighbors of process j have updated their sn values, process k initiates a stable wave by executing action $AR9$ (See Figure **??**).
- When the leader l receives the stable wave, the state of all processes in the system is stable. This completes the recovery of the system into $S1$ (See Figure **??**).

VI. Proof of Correctness for AR

In this section, we, first, explain different types of faults that may happen in the system and then proof the correctness of our proposed protocol.

A. Fault Types

We assume that the processes are in the presence of (a) fail-stop faults and (b) transient faults. If a process fail-stops, it cannot communicate with the other processes and a tree correction algorithm needs to reconfigure the tree. Transient faults can perturb all variables (e.g., sn, st, etc.) except $otsn$ and $ctsn$ values. The corruption of the $otsn$ and $ctsn$ values is discussed in Section **??**. Moreover, we assume that all the faults stop occurring after some time. We use f to denote these faults in the rest of this paper.

B. Proof of Correctness

To show the correctness of AR, we define the predicates T and AS, restoration state predicate $S2$, and invariant $S1$ in the following and use them for subsequent discussions.

$$S1 : \forall j,k : ((st.j = stable \vee st.j = restore) \wedge lc.j \wedge (P.j \text{ forms the tree}) \wedge (otsn.j = ctsn.k) \wedge l.j = \{\text{leader of the parent tree}\})$$
$$T : \forall j,k : otsn.j = ctsn.k \wedge (st.j = restore \vee st.j = stable)$$
$$AS : max(otsn.j) \geq max(ctsn.j)$$
$$S2 : \forall j,k : ((st.j = \top \vee st.j = \bot) \wedge (P.j = j \wedge st.j = \top \Rightarrow otsn.k \geq otsn.j))$$

In $S1$, the state of all processes is either $stable$ or $restore$ and all $otsn$ and $ctsn$ values are equal. If some faults occur but no auditable events, the system reaches T where all $otsn$ and $ctsn$ values are still equal. In this case, the state of the processes cannot be perturbed to $\bot$ or $\top$. If auditable events occur, the system goes to AS where the $otsn$ and $ctsn$ values are changed and the $max(otsn)$ is always greater than and equal to $max(ctsn)$. When all the states are changed to $\bot$ or $\top$, all processes are aware of the auditable events and the system is in $S2$. Also, the constraint $otsn.k \geq otsn.j$ in the definition of $S2$ means that when the state of the leader is $\top$, the rest of the processes are aware of the auditable events that has caused the leader to change its $otsn$.

Theorem 1: Upon starting at an arbitrary state in T, in the absence of faults and auditable events, the system is guaranteed to converge to a state in $S1$.

Proof: Since there is no auditable event in the system, action $AR1$ cannot execute and the $otsn$ and $ctsn$ values do not change. As a result, the system remains in T and executing actions $AR8$, $AR9$, and $AR10$ converges the system to $S1$. ■

Let f be the faults identified in Section **??**. We have:

Theorem 2: T is closed in actions $(AR2–AR11) \cup f$.

Proof: We assume that faults cannot perturb the $otsn$ and $ctsn$ values. Additionally, action $AR1$ does not execute to change the $otsn$ value. This guarantees that actions $AR2–AR6$ cannot execute to change the $otsn$ or $ctsn$ values. Hence, when a system is in T, in the absence of auditable events, it remains in T. Moreover, faults cannot perturb the system to a state outside of T, thereby closure of T. ■

Corollary 1: Upon starting at an arbitrary state in T, in the presence of faults but in the absence of auditable events, the system is guaranteed to converge to a state in $S1$.

Lemma 1: Starting from a state in T where the $otsn$ values of all processes equal x, if at least one auditable event occurs, the system reaches a state where $\forall j : otsn.j \geq x + 1$.

Proof: This lemma implies that, if there is an auditable event in the system and at least one process detects it and increments its $otsn$ value, eventually all processes will be aware of that auditable event.

When the system is in T, all the $otsn$ values are equal to x. After detecting an auditable event, a process increments its $otsn$ to $x + 1$ by executing action $AR1$. Consequently, using action $AR2$, every process gets notified of the auditable event and increments its $otsn$ value. If some other process detects more auditable events in the system, it increments its $otsn$ value and notifies the other processes of those auditable events. Hence, the $otsn$ value of all processes is at least $x + 1$. ■

Theorem 3: Starting from any state in $AS - T$ where $max(otsn.j) > max(ctsn.j)$ and the auditable events stop occurring, the system is guaranteed to reach a state in $S2$.

Proof: When there is a process whose $otsn$ value is greater than all the $ctsn$ values in the system, the state of the system is one of the following:
- there is at least one process that is not aware of all the auditable events occurred in the system, or
- all processes are aware of all auditable events occurred but their states are different.

The first case illustrates that all $otsn$ values are not equal. Consequently, action $AR2$ executes and makes all $otsn$ values equal. When the leader gets notified of the auditable events, it initializes a $\bot$ wave by executing action $AR3$. This wave propagates towards the leaves by executing action $AR4$. When a leaf receives the $\bot$ wave, it changes its state to $\top$ by executing action $AR5$ and propagates the $\top$ wave towards the leader. Note that, action $AR2$ executes concurrently with the other actions. Thus, all $otsn$ values will eventually become equal. The second case shows that all $otsn$ values are equal but the state of the leader is not changed to $\top$ yet. Therefore, when all $otsn$ values are equal and the auditable events stop occurring, the leader executes action $AR5$ and changes its state to $\top$, thereby reaching $S2$. Consider that, if a process fails and causes some changes in the configuration of the tree, the res variable of its neighbors would be reset to false and the leader will eventually get notified of this failure by executing action $AR5$. Thus, the leader initializes a new $\bot$ by executing action $AR5$. Moreover, if the leader fails when its state is $\bot$ and another process, say j, whose state is $\top$ becomes the new leader, the guard of action $AR2$ becomes true since $otsn.j > ctsn.j$. In this case, the new leader initializes a new $\bot$ wave to ensures that when the state of the leader is $\top$, the state of all the other processes in the system is also $\top$. ■

Theorem 4: Starting from a state in T and the occurrence of at least one auditable event, the system is guaranteed to reach $S2$ even if auditable events do not stop occurring.

Proof: occurring, at least, one auditable event, some process detects it and increments its $otsn$ by executing action $AR1$. Following Lemma **??**, all processes will eventually get notified of the auditable event. Hence, all $otsn$ values will be equal and, following Theorem **??**, the system will reach a state where the state of all processes is either $\top$ or $\bot$. Also if the state of the leader is $\top$, we can ensure that all processes are notified of the auditable event, thereby reaching a state in $S2$.

Note that, even if the auditable events continue occurring, the process states do not change and the system remains in $S2$. ■

Theorem 5: Starting from a state in T and in the presence of faults and auditable events, the system is guaranteed to converge to $S1$ provided that faults and auditable events stop occurring.

Proof: Following Theorem **??**, if faults and auditable events occur, the system reaches a state in $S2$ where all $otsn$ values are equal and the leader process is aware that all processes have been notified of the auditable events. In this situation, the $ctsn$ value of the leader is equal to its $otsn$ value (by executing the statement $ctsn.j := otsn.j$ in action $AR5$). Consequently, the leader can start a restoration wave by executing action $AR7$. Moreover, the other processes can concurrently execute action $AR6$, increase their $ctsn$ value, and propagate the restoration wave by executing action $AR8$. When all $ctsn$ values are equal, the system is in T and following Theorems **??** and **??**, the system converges to $S1$. ■

Observation 1: The system is guaranteed to recover to $S2$ in the presence of faults and auditable events. As long as auditable events continue occurring, the system remains in $S2$ showing that all processes are aware of the auditable events. When the auditable events stop occurring, the system converges to $S1$, where the system continues its normal execution.

VII. The Corruption of AR Variables

Protocol AR is designed with the assumption that auditable events arise due to conflicting system requirements, dynamic adaptation, tampering etc. Hence, these events are expected to be rare since there is a *cost* (e.g., legal) associated with creating/causing these events. It does not consider the case where the adversary intentionally tries to affect variables of AR simply to reduce performance. In this section, we show how AR can be revised so that perturbation by adversary in this manner can be managed. As a side effect, it also demonstrates how AR can manage potential corruption of $otsn$ and $ctsn$ variables. Since one interacts with protocol AR by changing $otsn$ and/or $ctsn$ variables, next, we discuss the effect of corrupting these variables (either by transient faults or maliciously). Specifically, first, in Section **??**, we discuss how one can prevent and/or manage corruption of $otsn$ values. Then, in Section **??**, we identify how one can prevent and/or manage corruption of $ctsn$ values.

A. Preventing and Managing $otsn$ Corruption

The variable $otsn$ may get corrupted by transient faults and its value can be (1) increased or (2) decreased to an incorrect value. Next, we explain what would happen in each case and how to prevent such corruptions.

Managing incorrect increase of $otsn$. If the $otsn$ value of some process gets increased incorrectly, this would result in a false alarm. In other words, this increase would be treated as a new auditable event. Hence, the system would go into $S2$. In turn, the leader process would issue a new reset action resulting in the system being restored to normal states. Thus, increasing $otsn$ does not endanger the system or cause violation of system requirements, but it can reduce the system performance. (If auditable events at different processes are correlated then the protocol can be modified so that a process propagates an auditable event only if it obtains multiple independent notifications of the auditable events. This could be achieved by requiring process j to maintain $otsn.j$ to be an array instead of just a scalar as done in Protocol AR. When j maintains such an array, $otsn.j[k]$ would denote the number of auditable events originated by k that j is aware of. And, the leader process will execute action AR3 only if the number of processes notifying it of auditable events exceeds a given threshold.) To prevent this corruption, we can add sufficient redundancies into the program similar to that in [**?**]. In particular, one can maintain several copies of the $otsn.j$ (possibly on processes other than j as well) and j can use a simple majority to identify its $otsn$ value. This will increase the work involved in increasing the $otsn$ value. However, since auditable events are rare, the impact will be negligible.

Prevent incorrect decrease of $otsn$. If the $otsn$ value of some process is decreased incorrectly then this would create an undesired outcome. For instance, consider the case where the $otsn$ value is decreased by 1. As a result, if auditable events are detected, incrementing $otsn$ will remain hidden and the other processes will not get notified of the auditable events and the auditable events will be lost. To prevent this corruption, we can use the idea of *append-only log* (e.g., [**?**]). Using this idea, we need to separate the storage of $otsn$ values from the index used to locate the values. In particular, $otsn$ values are stored in an append-only log on a RAID array of disk drives. The simplicity of the append-only log structure eliminates many possible software errors that might cause data corruption and facilitates a variety of additional integrity strategies. A separate index structure allows the $otsn$ values to be efficiently located in the log. However, the index can be regenerated from the data log if required and thus does not have the same reliability constraints as the log itself. With such append-only logs, one can obtain the most recent $otsn$ value from the log preventing any impact of a transient fault that causes $otsn$ value to decrease. Finally, we also note that both the approaches for dealing with reduction of $otsn$ and increase in $otsn$ are compatible with each other.

B. Preventing and Managing $ctsn$ Corruption

There are subtle differences between the effect of $otsn$ corruption and that of $ctsn$ corruption. Hence, we need a different approach for managing undesired changes to $ctsn$. An increase in $otsn$ due to the corruption resulted in unnecessary restoration to auditable state and unnecessary restoration to the set of legitimate states. By contrast, an increase in $ctsn.j$ value due to corruption will cause process j to believe that more auditable events have been cleared by the leader process. Essentially, this will cause some auditable events to be lost. To avoid this, we can utilize the fact that only the leader process is allowed to increment $ctsn$ value by itself. Others only rely on $ctsn$ values of other process to increase their own value. Hence, we can simply use encryption approach to prevent such corruption. In particular, we can assign a public/private key pair for the leader process. And, instead of maintaining $ctsn$ as an integer value, for each non-leader process, we can encrypt it by the private key of the leader process. Thus, it is highly unlikely that the $ctsn$ value gets changed to another valid $ctsn$ value. Instead, a transient fault will only result in the $ctsn$ value to be corrupted in an easily detectable manner. Hence, other processes can easily ignore it. Moreover, the corrupted process can be easily corrected by action $AR6$. Finally, this decryption is needed only when a process changes its $ctsn$ value. Checking if $ctsn$ values of two processes are identical can be achieved by comparing the encrypted values.

Additionally, to manage the corruption of $ctsn$ value of the leader, we can utilize the same approach that we used for $otsn$. In particular, we could use append-only log so the leader can obtain the last $ctsn$ value it had from the log. This approach would require that the adversary cannot append log with 'encrypted' content. Adversary could, however, append to append-only logs. But if it does not correspond to valid $ctsn$ value, the leader process can simply ignore it. Alternatively, we can keep multiple copies of $ctsn$ in a fashion similar to that for handling $otsn$. This would require an assumption that adversary cannot corrupt all of these copies.

VIII. Stabilizing Auditable Restoration Protocol with Bounded State Space

The protocol AR presented in Section **??** assumes that the $otsn$, $ctsn$, and sn values are unbounded. And, the stabilization property relied on this assumption. It is well-known that combining stabilization and bounded state space is a difficult problem [**?**]. The main intuition is that in stabilization, recovery from any arbitrary state is required. Hence, if the value is perturbed to the max value, providing recovery is difficult/impossible. In this section, we present a new protocol, BAR, so that these variables are bounded while still preserving stabilization. The changes in this section also address how one can deal with a process that reports auditable events maliciously or due to errors in the mechanism used to identify auditable events.

A. Bounding $otsn$

In the protocol in Figure **??**, $otsn$ values continue to increase in an unbounded fashion as the number of auditable events increases. In that protocol, if $otsn$ values are bounded and eventually restored to 0 upon reaching the bound, then this may cause the system to lose some auditable events. Furthermore, if $otsn.j$ is reset to 0 but it has a neighbor k where $otsn.k$ is non-zero, it would cause $otsn.j$ to increase again.

Before we present our approach, we observe that if the auditable events are too frequent, restoring the system to legitimate states may never occur. This is due to the fact that if the authorized process attempts to restore the system to legitimate states then its action would be *canceled* by new auditable events. Hence, restoring the system to legitimate states can occur only after auditable events stop. However, if auditable events occur too frequently for some duration, we want to ensure that the $otsn$ values still remain bounded.

Our approach is as follows: We change the domain of $otsn$ to be N^2+1 ($0 \cdots N^2$), where N is the number of processes in the system. Furthermore, we change action $AR1$ such that process j ignores the detectable events if its neighbors are not aware of the recent auditable events that it had detected. Essentially, in this case, j is *consolidating* the auditable events. This is acceptable since consolidating happens only if previous events have not been handled when new auditable event happens. Furthermore, we change action $AR2$ by which process j detects that process k has detected a new auditable event. In particular, process j concludes that process k has identified a new auditable event if $otsn.k$ is in the range $[otsn.j \oplus 1 \cdots otsn.j \oplus N]$, where $\oplus$ is modulo N^2+1 addition. Moreover, before j acts on this new auditable event, it checks that its other neighbors have caught up with j, i.e., their $otsn$ value is in the range $[otsn.j \cdots otsn.j \oplus N]$. Essentially, this action ensures that process j determines a higher value of its neighbor to be a new auditable event only if the $otsn$ values of the neighbors are *reasonably ahead* of it. Finally, the third action takes care of the case where $otsn$ values of neighboring processes are *unreasonable*. In particular, we add another action where $otsn.j$ and $otsn.k$ are far apart, i.e., $otsn.j$ is not in the range $[otsn.k \ominus N \cdots otsn.k \oplus N]$. Thus, the revised and new actions are $BAR1$, $BAR2$, and $BAR12$ in Figure **??**.

We have utilized these specific actions so that we can benefit from previous work on asynchronous unison [**?**] that is designed to ensure that eventually clock of any two neighboring processes differ by at most 1. Although the protocol in [**?**] is designed for clock synchronization, we can use its proof of correctness to bound the $otsn$ values. In particular, the above actions are same (except for the detection of new auditable events) as those of [**?**]. (For sake of completeness, we give the actions of the protocol in [**?**] in Appendix.) Hence, based on the results from [**?**], we can observe that if some process continues to detect auditable events forever, eventually, the system would converge to a state where the $otsn$ values of any two neighboring processes differ by at most 1. Thus, we have

Theorem 6: Starting from an arbitrary state, even if the auditable events occur at any frequency, the program in Figure **??** converges to states where for any two neighbors j and k: $otsn.j = otsn.k \ominus 1$, $otsn.j = otsn.k$, or $otsn.j = otsn.k \oplus 1$.

Proof: Proof follows from [**?**]. ■

Next, we consider what happens to the revised protocol when auditable events stop occurring. In particular, we have

Theorem 7: Starting from an arbitrary state, if the auditable events stop occurring, the program in Figure **??** converges to states where for any two neighbors j and k: $otsn.j = otsn.k$.

Proof: As we mentioned in Theorem **??**, the system is guaranteed to converge to a state where for any two neighbors j and k either $otsn.j = otsn.k \ominus 1$, $otsn.j = otsn.k$, or $otsn.j = otsn.k \oplus 1$. Now, $otsn$ values of any two processes (even if they are not neighbors) differ by at most $N-1$. In other words, there exists a and b such that for some processes j and k, $otsn.j = a$ and $otsn.k = b$, where b is in the range $[a \cdots a \oplus (N-1)]$ and the $otsn$ values of remaining processes are in the range $[a \cdots b]$.

Hence, processes are not far apart each other and the action $BAR12$ cannot execute. In addition, action $BAR1$ cannot execute since the auditable events have stopped occurring. Hence, by executing action $BAR2$, each process increases its $otsn$ such that the $otsn$ values will be equal to b for all processes. ■

From this theorem, we have

Corollary 2: Under the assumption that a process does not detect new auditable event until it is restored to legitimate states, we can guarantee that $otsn$ values will differ by no more than 1.

Finally, observe that with the above change in program actions and Theorem **??**, stabilization property is preserved even if $otsn$ is bounded in the manner discussed above.

B. Bounding $ctsn$

The above approach bounds the $otsn$ value. However, the same approach cannot be used to bound $ctsn$ value. This is due to the fact that the value to which $ctsn$ converges may not be related to the value that $otsn$ converges to. This is unacceptable and, hence, we use the following approach to bound $ctsn$.

BAR1 :: $\{j$ *detects an auditable event* $\}$
$\forall k : otsn.k \in [otsn.j \cdots otsn.j \oplus N]$
$\longrightarrow otsn.j := otsn.j \oplus 1$

BAR2 :: $\forall k : otsn.k \in [otsn.j \cdots otsn.j \oplus N] \wedge$
$\exists k : otsn.k \in [otsn.j \oplus 1 \cdots otsn.j \oplus N]$
$\longrightarrow otsn.j := otsn.j \oplus 1$

BAR3 :: $P.j = j \wedge st.j \neq \perp \wedge \ otsn.j \neq ctsn.j$
$\longrightarrow st.j, sn.j, res.j := \perp, sn.j + 1, min(res.j + 1, 1)$

BAR4 :: $st.(P.j) = \perp \wedge sn.j \neq sn.(P.j) \wedge l.j = l.(P.j)$
$\longrightarrow st.j, sn.j, res.j := \perp, sn.(P.j), res.(P.j)$

BAR5 :: **AR5**
BAR6 :: $ctsn.j \neq ctsn.(P.j) \longrightarrow ctsn.j := ctsn.(P.j)$
BAR7 :: **AR7**
BAR8 :: $st.(P.j) = restore \wedge sn.j \neq sn.(P.j) \wedge$
$l.j = l.k \wedge otsn.j = ctsn.j$
$\longrightarrow st.j, sn.j, res.j := restore, sn.(P.j), res.(P.j)$

BAR9–BAR10 :: **AR9–AR10**
BAR11 :: $\langle$ *any tree correction action that affects process* j $\rangle$
$\longrightarrow res.j := -1$
BAR12 :: $(otsn.j \notin [otsn.k \ominus N \cdots otsn.k \oplus N]) \wedge$
$(otsn.j > otsn.k) \longrightarrow otsn.j := 0$

Fig. 4: Bounded auditable restoration protocol, BAR.

First, in action $AR6$, the guard $ctsn.j > ctsn.k$ needs to be replaced by $ctsn.j \neq ctsn.(P.j)$. Thus, the new action $BAR6$ is shown in Figure **??**.

Second, we require to replace the notion of *greater than* by *not equal* in all the actions of Figure **??** since we are bounding the $otsn$ and $ctsn$ values. Hence, we change action $AR3$ to $BAR3$ in the program in Figure **??**.

With these changes, starting from an arbitrary state, after the auditable events stop, the system will eventually reach states where all $otsn$ values are equal (cf. Theorem **??**). Subsequently, if $otsn$ and $ctsn$ values of the leader process are different, it will execute action $AR7$ to restore the system to an auditable state. Then, the leader process will reset its $ctsn$ value to be equal to $otsn$ value. Finally, these values will be copied by other processes using action $BAR6$. Hence, eventually all $ctsn$ values will be equal.

Theorem 8: Starting from an arbitrary state, if the auditable events stop occurring, the program in Figure **??** reaches states where for any two neighbors j and k: $otsn.j = otsn.k = ctsn.j = ctsn.k$.

Proof: According to Theorem **??**, all $otsn$ values will eventually be equal. Moreover, when the leader ensures that all processes are aware of the auditable events, it executes action $AR5$ and updates its $ctsn$ value by its $otsn$ value. Consequently, when the rest of processes detect this change, they execute action $BAR7$ and update their $ctsn$ values. Hence, eventually all $ctsn$ values will be equal. ■

Finally, we observe that even with these changes if a single auditable event occurs in a legitimate state (where all $otsn$ and $ctsn$ values are equal) then the system would reach a state in the auditable states, i.e., Theorem **??** still holds true with this change. In other words, stabilization property is preserved even if $otsn$ and $ctsn$ are bounded in the manner discussed above.

C. Bounding sn

Our goal in bounding sn is to only maintain $sn \ mod \ 2$ with some additional changes. To identify these changes, first, we make some observations about how sn values might change during the computation in the presence of faults such as process failure but in the absence of transient faults.

Now, consider the case where we begin with a legitimate state of the auditable restoration protocol where all sn values are equal to x. At this time, if the leader process executes actions $AR3$ or $AR7$ then sn value of the leader process will be set to $x + 1$. Now, consider the sn values of processes on any path from the leader process to a leaf process. It is straightforward to observe that some initial processes on this path will have the sn value equal to $x + 1$ and the rest of the processes on this path (possibly none) will have the sn value equal to x. Even if some processes fail, this property would be preserved in the part of the tree that is still connected to the leader process. However, if some of the processes in the subtree of the failed process (re)join the tree, this property may be violated.

Thus, the structure of a path from the leader process to a leaf process is as shown in Figure **??**. Let j denote the first *newly joined* process on this path. As shown in this figure, if the sn value of the leader l is $x + 1$ then some ancestors of j have sn value set to x and some ancestors have the sn value set to $x + 1$. Even if we maintain only modulo 2 information for sn values, all ancestors of j (excluding j) can differentiate between the values x and $x + 1$. However, since $sn.j$ may not be related to x or $x + 1$, it is possible that $sn.j \ mod \ 2$ is same as $x \ mod \ 2$ or $(x+1) \ mod \ 2$. Therefore, if we only maintain modulo 2 information for sn then j may not be able to determine if its sequence number is the same as that of the leader.

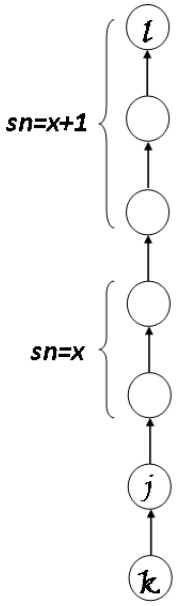


Fig. 5: An observation about how sn values may change.

To deal with this situation, we allow this newly joined process to participate in the auditable and restoration waves by executing actions $AR4$ and $AR8$. When it does so, it gets the sequence number from its parent. Moreover, from the above discussion, the sequence number of the parent is either x or $x+1$. However, we force j to *fail* this wave by settings its result to false. In turn, the leader will increment the sn and start a new wave with sequence number $x+2$. At this point, j would have a sequence number that is consistent with that of the leader. Hence, when the leader starts a new auditable or reset wave with sequence number $x+2$, j can conclude that its sequence number is same as that of the leader even if j is only maintaining $sn.j \ mod \ 2$. Also, if j has a child, say k (cf. Figure **??**), then when k propagates the new wave, it should also abort the first two waves. Based on the above discussion, we change actions $AR3$, $AR4$, $AR8$, and $AR11$ by actions $BAR3$, $BAR4$, $BAR8$, and $BAR11$ in the program in Figure **??**.

Theorem 9: The space maintained by BAR is $O(log \ N)$, where N is the number of processes in the system.

Proof: The space needed for variables sn, res, and st in BAR are 1, 1, and 2 bit(s), respectively. Moreover, the domain of $otsn$ and $ctsn$ is $N^2 + 1$. Thus, we need $log(N^2 + 1) \approx 2(log\ N)$ space for maintaining $otsn$ and $ctsn$. Therefore, the space maintained by BAR is $O(log\ N)$. ■

D. Analyzing the Program after Bounding $otsn$, $ctsn$ and sn

In this section, we show that after bounding $otsn$, $ctsn$ and sn as discussed above, the resulting program is stabilizing fault-tolerant. We also show that if the resulting program is perturbed by faults such as failstop faults, then it restores to legitimate states. Moreover, during this recovery, no auditable events are lost. In other words, if the program starts in a legitimate state and is perturbed by an auditable event then the system will eventually reach an auditable state even if faults such as failstop faults occur. Finally, we also discuss possible extensions if an unbounded number of auditable events are permitted to occur.

Preservation of stabilization property. In this section, we argue that the stabilization property is preserved even if $otsn$, $ctsn$ and sn are bounded as discussed above. Specifically, after faults and auditable events stop, eventually all $otsn$ values become equal. At this point, if $ctsn$ value of the leader is different from its $otsn$ value, the system will be restored to the auditable state. Subsequently, the $ctsn$ value of the leader will be the same as its $otsn$ value. Since this value is copied by all neighbors (via action $BAR2$), it would cause the system to reach a state where all $otsn$ values will be equal. Moreover, action $BAR8$ ensures that in the absence of new waves initiated in actions $BAR3$ and $BAR7$, all sequence numbers are equal. Thus,

Theorem 10: Auditable restoration program is stabilizing, i.e., starting from an arbitrary state in T, after auditable events and faults stop, the program converges to $S1$, where

$S1 : \forall j, k : ((st.j = stable \vee st.j = restore) \wedge lc.j \wedge$
$(P.j$ *forms the tree*$) \wedge\ (otsn.j = ctsn.k) \wedge$
$l.j = \{$*leader of the parent tree*$\})$

Proof: Following Theorem **??** and explanations above, after auditable events and faults stop, the program converges to $S1$. ■

Analysis from legitimate states in the presence of faults and/or auditable events. In the program obtained after bounding $otsn$, $ctsn$ and sn, consider the case where one auditable event occurs in a legitimate state (i.e., in $S1$) and no other auditable event occurs until the system is restored to the legitimate states (i.e., in $S2$). Next, we show that this auditable event will cause the system to reach an auditable state. And, subsequently, it will be restored to a legitimate state.

In this case, for any two processes j and k, $otsn.k$ will be either $otsn.j$ or $otsn.j \oplus 1$. To show the desired property, we redefine predicates $S2$ and AS in the following. The definitions of $S1$ and T remain unchanged.

$AS' : \forall j, k : otsn.k \in [max(ctsn.j) \cdots max(ctsn.j) \oplus 1]$

$S2' : \forall j, k : ((st.j = \top \vee st.j = \bot) \wedge$
$(P.j = j \wedge st.j = \top \Rightarrow otsn.k \in [otsn.j \cdots otsn.j \oplus N])$

Thus, we have

Theorem 11: Starting from an arbitrary state in T, if exactly one auditable event occurs, the system is guaranteed to reach a state in $S2$ and then converge to $S1$ provided that faults stop occurring. Moreover, the states reached in such computation are a subset of AS'.

Proof: If exactly one auditable event occurs, the system is guaranteed to reach $S2$ and then it converges to $S1$ following the explanations above. ■

We can easily extend the above theorem to allow upto N auditable events before the system is restored to its set of legitimate states. The choice of N indicates that each process detects the auditable event at most once before the system is restored to its legitimate states. In other words, a process ignores auditable events after it has detected one and the system has not been restored corresponding to that event. In this case, the constraint AS' above needs to be changed to:

$AS'' : \forall j, k : otsn.k \in [max(ctsn.j) \cdots max(ctsn.j) \oplus N]$

Theorem 12: Starting from an arbitrary state in T, if upto N auditable events occur, the system is guaranteed to reach a state in $S2$ and then converge to $S1$ provided that faults stop occurring. Moreover, the states reached in such computation are a subset of AS''.

Proof: Since each process detects at most one auditable event before the restoration, executing actions $BAR1$ and $BAR2$, all $otsn$ values will become equal and the system reaches $S2$. Consequently, the leader initialize a restoration wave and the system converges to $S1$. ■

Remark. Observe that in the above two theorems, we consider the case where we start from any state in T, where faults such as failstop faults have occurred and could occur. In other words, reaching an auditable state (respectively, legitimate states) will occur even if it is perturbed by (finite number of) faults.

Dealing with Unbounded number of auditable events. From Theorem **??**, we observe that upto N auditable events will still cause the system to recover to an auditable state. Thus, the natural question is what happens if the number of auditable events is larger.

If the domain of $otsn$ is bounded, it is potentially possible that j starts from a state where $otsn.j$ equals x and there are enough auditable events so that the value of $otsn.j$ rolls over back to x. In our algorithm, in this case, some auditable events may be lost. We believe that this would be acceptable for many applications since the number of auditable events being so high is highly unlikely. Also, the domain of $otsn$ and $ctsn$ values can be increased to reduce this problem further. This problem can also be resolved by ensuring that the number of events detected by a process within a given time-span is bounded by allowing the process to ignore frequent auditable events.

We note that theoretically the above assumption is not required. The basic idea for dealing with this is as follows: Each process maintains a bit $changed.j$ that is set to true whenever $otsn$ value changes. Hence, even if $otsn$ value rolls over to the initial value, $changed.j$ would still be true. It would be used to execute action $BAR3$ so that the system would be restored to $S2$ and then to $S1$. In this case, however, another computation would be required to reset $changed.j$ back to false. The details of this protocol are outside the scope of this paper.

IX. Applications of Auditable Restoration Protocol

In this section, we discuss the applications of our auditable restoration protocol. The auditable restoration protocol is intended to be combined with applications where the auditable restoration protocol is a subsystem. We discuss how the auditable restoration subsystem interacts with the application in Section **??**. Subsequently, we illustrate this structure with an application in Section **??**.

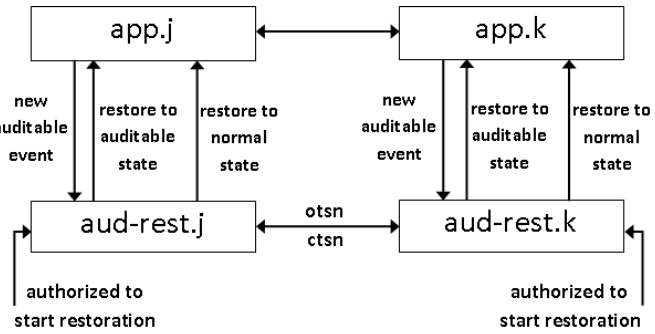


Fig. 6: The structure of auditable restoration subsystem.

A. Structure of auditable restoration protocol

Similar to the structure from [**?**], we envision the auditable restoration protocol to be used in systems where each process j in the system consists of two modules: an *application* module (represented by $app.j$ in Figure **??**) and an *auditable restoration* module (represented by *aud-rest.j* in Figure **??**). The task of module $app.j$ is application specific. The application modules of all processes constitute the desired goal of the system. An instance of these applications is described in the next section. The task of *aud-rest.j* is to provide the ability to reset the system to auditable and/or legitimate states. Hence, it maintains variables (such as $otsn.j$, $ctsn.j$, $sn.j$, etc.) maintained by the protocol in Section **??**.

The application module provides a mechanism by which it notifies the auditable restoration protocol of a new auditable event. In turn, the auditable restoration protocol will increase the value of $otsn.j$ and possibly restore the system to the auditable state thereafter. As part of the auditable restoration, this auditable event would be communicated to the auditable restoration modules on other processes. Also, in actions $AR2$, $AR3$, $AR7$, and $AR8$, the restoration protocol needs to restore the system to either the auditable or the normal states. Hence, the application module needs to provide an interface so that the application module at process j can be reset by the auditable restoration module at process j. Finally, each module *aud-rest.j* at process j has an input that shows if process j is an authorized process. By setting this input, the user authorizes the processes that can start the restoration wave.

B. Mail/Banking/Shopping Applications

Several mail/banking/shopping services require a two-factor authentication where the user first enters the password on a website and is informed of a one-time use code via text message. The user needs to enter this one-time code before giving system access. From a single user perspective, this may appear to be a centralized system. However, in reality, it is a distributed system due to different services offered and for replication.

Nodes in the auditable restoration system. The nodes considered in this system are the nodes that are responsible for providing the user with necessary services associated with mail/banking/shopping, etc. The application modules in this system communicate with each other so the user can obtain the desired service. In the following discussion, we assume that any changes made to this system affect only the corresponding users. If these nodes are used by other users in the system, changes done on behalf of one user do not affect the other users. (Intuitively, this would correspond to implementing the auditable restoration protocol per user.)

Need for auditable restoration and the existence of auditable events. In some cases, 2-factor authentication is not possible. Causes for this may include travel outside the coverage area, lack of cell service, etc. To avoid disruption of service, users are provided with a list of one-time tokens that can be carried with the users. The user can use one of the tokens to access the service. Inherently, this reduces the level of security provided to the user since these one-time tokens could be lost/stolen, etc. Moreover, the use of one-time password is different than using 2-factor authentication. Hence, whenever, the user uses a one-time token, this can be viewed as an auditable event.

This auditable event will be visible to the nodes that were responsible for the user logging into the system, but it may not be visible to other nodes that the user may obtain request from. The auditable restoration protocol can be used to ensure that these nodes are aware of the latest auditable event even in the presence of faults and additional auditable events.

Role of auditable states. When a node is notified of the auditable event, as part of auditable restoration, in action $AR4$, the node will restore itself to an auditable state. In the context of these applications, an auditable state may allow the user a limited services and block some sensitive services. For example, in case of a shopping service, the user may be blocked from using saved credit cards. In case of a banking system, transactions may be limited to a certain monetary values. As an example for mail servers, the functionality of the user's account may be decreased such that the user cannot utilize his account to log into any account other than his mail account (e.g. bank accounts, file sharing accounts, etc).

Role of normal states. When the user logs in later and provides a 2-factor authentication, we need to *clear* any restrictions placed on the user in the auditable state. The authentication node can act as the leader process in action $AR7$. As part of the restoration protocol, each node will be notified to restore to the normal state.

Interaction between auditable events and the restoration wave. In the event of leakage of the one-time codes, an adversary could use them to log into the account. However, the use of auditable restoration will prevent the user from accessing most sensitive services until a 2-factor authentication by user is provided. At this point, the user could deactivate the one-time codes if it is determined that they were compromised. Moreover, even without this, if the adversary uses another one-time code, this will be a new auditable event, thereby ensuring that the system reaches an auditable state once again.

X. Related Work

Stabilizing Systems. There are numerous approaches for recovering a program to its set of legitimate states. Arora and Gouda's distributed reset technique introduced in [**?**] is directly related to our work and ensures that after completing the reset, every process in the system is in its legitimate states. However, their work does not cover faults (e.g., process failure) during the reset process. In [**?**], we extended the distributed reset technique so that if the reset process is initialized and some faults occur, the reset process works correctly. In [**?**], Katz and Perry showed a method called global checking and correction to periodically do a snapshot of the system and reset the computation if a global inconsistency is detected. This method applies to several asynchronous protocols to convert

them into their stabilizing equivalent, but is rather expensive and insufficient both in time and space.

Moreover, in nonmasking fault-tolerance (e.g., [**?**], [**?**]), we have the notion of fault-span too (similar to T in Definition **??**) from where recovery to the invariant is provided. Also, in nonmasking fault-tolerance, if the program goes to $AS-T$, it *may* recover to T. By contrast, in auditable restoration, if the program reaches a state in $AS-T$, it is required that it first restores to $S2$ and then to $S1$. Hence, auditable restoration is stronger than the notion of nonmasking fault-tolerance.

Auditable restoration can be considered as a special case of nonmasking-failsafe multitolerance (e.g., [**?**]), where a program that is subject to two types of faults F_f and F_n provides (i) failsafe fault tolerance when F_f occurs, (ii) nonmasking tolerance in the presence of F_n, and (iii) no guarantees if both F_f and F_n occur in the same computation.

In the context of Cyber-Physical Systems (CPS), in [**?**], the authors propose a recovery mechanism to maintain the stability of the distributed CPS when the communication is lost. Nevertheless, they only limit their technique to communication loss and do not consider any other types of faults nor auditable events.

Tamper-Evident Systems. These systems [**?**], [**?**], [**?**] use an architecture to protect the program from external software/hardware attacks. An example of such architecture, AEGIS [**?**], relies on a single processor chip and can be used to satisfy both integrity and confidentiality properties of an application. AEGIS is designed to protect a program from external software and physical attacks, but did not provide any protection against side-channel or covert-channel attacks. In AEGIS, there is a notion of recovery in the presence of a security intruder where the system recovers to a 'less useful' state where it declares that the current operation cannot be completed due to security attacks. However, the notion of fault-tolerance is not considered. In the context of Figure **??**, in AEGIS, $AS-T$ equals $\neg S1$, and $S2$ corresponds to the case where tampering has been detected.

Byzantine-Tolerant Systems. The notion of Byzantine faults [**?**] has been studied in a great deal in the context of fault-tolerant systems. Byzantine faults capture the notion of a malicious user being part of the system. Typically, Byzantine fault is mitigated by having several replicas and assuming that the number of malicious replicas is less than a threshold (typically, less than $\frac{1}{3}$rd of the total replicas). Compared with Figure **??**, T captures the states reached by Byzantine replicas. However, no guarantees are provided outside T.

Byzantine-Stabilizing Systems. The notion of Byzantine faults and stabilization have been combined in [**?**], [**?**], [**?**]. In these systems, as long as the number of Byzantine faults is below a threshold, the system provides the desired functionality. In the event the number of Byzantine processes increases beyond the threshold temporarily, the system eventually recovers to legitimate state. Similar to systems that tolerate Byzantine faults, these systems only tolerate a specific malicious behavior performed by the adversary. It does not address active attacks similar to that permitted by Dolev-Yao attacker [**?**].

In all the aforementioned methods, the goal is to restore the system to the legitimate states. In our work, if some auditable events occur, we do not recover the system to the legitimate states. Instead, we recover the system to restoration state where all processes are aware of the auditable events occurred. Moreover, in [**?**], if a process requests a new reset wave while the last reset wave is still in process, the new reset will be ignored. Nevertheless, in our work, we cannot ignore auditable events and all processes will eventually be aware of these events and the system remains in the auditable state as long as the auditable events continue occurring. Also the aforementioned techniques cannot be used to bound $otsn$ value in our protocol.

XI. Conclusion and Future Work

In this paper, we proposed two protocols that focused on the problem of auditable restoration. The first protocol has an unbounded state space while the second protocol utilizes a bounded state space that does not increase with the length of computation. To the best of our knowledge, this is the first protocol to achieve self-stabilization in bounded state space for auditable restoration. Bounding of auditable restoration variables, $otsn$ and $ctsn$ was challenging and required different approaches. Our key insight for bounding them was to identify a potential application of clock synchronization algorithm.

Both of our protocols provide a mechanism to deal with auditable events that allow one to capture issues such as tampering, conflicting requirements, and dynamic adaptation. The notion of auditable events allows us to capture *what happens if a tamper-resistant system is tampered with*. Specifically, systems often provide a mechanism such as *In-case-of-emergency-break-glass* where the expectation is that this mechanism will be used only in case of emergency because of some threat (e.g., legal). It also provides a mechanism for dealing with situations such as *Authorized access only-push door to open-alarm will ring*. In these situations, it is expected that providing some temporary unauthorized access is necessary for *greater good*. Although such situations arise frequently in computational and physical domain, ad-hoc approaches are necessary to deal with the case where such mechanisms are deployed. We argue that the system could be designed to have multiple modes where the effect of such auditable events would be to restore the system to a mode where every node is aware of the existence of these auditable events and take the mitigating action as necessary. By contrast, when such auditable events do not exist (or are cleared by an authorized process), the system can improve performance and security based on the assumption that none of its components/processes is tampered with. Thus, in case of the need for such temporary unauthorized access, auditable restoration provides a middle-ground between two extremes– disallow temporary unauthorized access and allow complete unauthorized access. Instead, it provides the ability to allow temporary unauthorized access but limit the level of access.

The notion of auditable events also allows one to deal with conflicting requirements. Examples of such requirements include cases where access must be restricted but in some access entirely preventing access is less desirable than some unauthorized access.

The notion of auditable events is also applicable in the design of adaptive systems. Specifically, in these contexts, we can have a monitor that checks the current state of the system. If the system is outside T_1 (cf. Figure **??**), it can invoke the restoration mechanism to move the system into S_2 where it provides a different tradeoff available to the adaptive system. Examples of such systems include intrusion detection systems where being outside T_1 may correspond to excessive traffic related to intrusions. And S_2 will correspond to states where the system takes actions that reduce the availability in order to protect the system. A similar approach is also used in many

systems to deal with unanticipated faults like the design of *MGS spacecraft* and several other systems [**?**]. Specifically, in these cases, the occurrence of such faults cause the system to be outside T_1. After detecting this, the system reaches a state where it maintains only minimal activities and waits for the operator to provide corrected instructions that will take care of the unanticipated situation. The state predicate S_2 corresponds to these states. And, the authorized entity corresponds to some component in the system that validates the corrected instructions received from the operator.

Our program guarantees that after auditable events occur the program is guaranteed to reach an auditable state where all processes are aware of the auditable event and the authorized process is aware of this and can initiate the restoration (a.k.a. clearing) operation. The recovery to auditable state is guaranteed even if it is perturbed by finite number of auditable events or faults. It also guarantees that no process can begin the task of restoration until recovery to auditable states is complete. Moreover, after the authorized process begins the restoration operation, it is guaranteed to complete even if it is perturbed by a finite number of faults. However, it will be aborted if it is perturbed by new auditable events.

Our program is stabilizing in that starting from an arbitrary state, the program is guaranteed to reach a state from where future auditable events will be handled correctly. It also utilizes only finite states, i.e., the values of all variables involved in it are bounded.

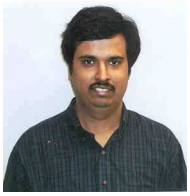
We are currently investigating the design and analysis of auditable restoration of System-on-Chip (SoC) systems in the context of the IEEE SystemC language. Our objective here is to design systems that facilitate reasoning about what they do and what they do not do in the presence of auditable events. Second, we plan to study the application of auditable restoration in game theory (and vice versa).

Acknowledgements. This work is supported by by NSF CNS 1329807, NSF CNS 1318678, and XPS 1533802.

XII. Biography

Reza Hajisheykhi is a PhD candidate in Computer Science and Engineering Department at Michigan State University. He received his M.Sc. in Computer Engineering from Sharif University of Technology in 2009. Currently, he is working in Software Engineering and Network Systems Laboratory (SENS Lab) under supervision of Dr. Sandeep Kulkarni.

Mohammad Roohitavaf is a PhD student in Computer Science and Engineering Department at Michigan State University. He received his M.Sc. in Computer Science from Sharif University of Technology in 2013. Currently, He is working in Software Engineering and Network Systems Laboratory (SENS Lab) under supervision of Dr. Sandeep Kulkarni.



Sandeep Kularni received PhD degree from the Ohio State university in 1999. After that, he joined Michigan State University. Currently he is a professor at the Department of Computer Science and Engineering at Michigan State University. His interests lie in Operating Systems, Distributed Systems, and Fault Tolerance.

Appendices

Appendix A
Discussion

What happens if the set of relevant processes changes depending on the auditable event? Does protocol AR change? No, protocol AR does not change. However, the presentation will become more complex. Specifically, it is possible that the exact set of relevant processes may depend upon the specific auditable event. Or, the tree algorithm supporting the auditable restoration may include all processes (this could happen if the tree is already formed for some other purpose in the protocol and we want to simply reuse the protocol rather than build a separate tree for reset). Essentially, in this case, all processes would participate in the restoration protocol even though they simply update the sn, $otsn$, etc., but do not actually reset their state. Alternatively, one would need to build a separate tree for each set of relevant processes. Once again, we feel that this is more about engineering the protocol for a given application and, hence, we have not considered it here for a general algorithm.

What happens if the leader changes/fails while protocol AR utilizes encryption to prevent/manage $ctsn$ corruption? The tree protocol considered in this paper [**?**] allows each process to be aware of the leader process. Hence, while using encryption for protecting $ctsn$, each process can also maintain the leader that assigned that value. Under normal circumstances (i.e., no auditable events), the $ctsn$ values are equal. Hence, even if the leader changes, each process would have to update the encrypted value of $ctsn$ (with the new leader). But since the actual $ctsn$ value does not change, there will be no other effect. If leader changes while restoration is ongoing, it is possible that the new leader has an older $ctsn$ value. This can happen if the old leader updates the $ctsn$ value using action $AR5$ but fails before this value is known to the new leader. In this case, the new leader would cause the system to be restored to auditable state once more before restoration to legitimate states is provided. However, this case is essentially the same as if the failure occurred during the restoration process, which is also handled by redoing the restoration phase.

How well does the append-only logs proposed in Section ?? fit the revised protocol from Section ??? One can utilize the append-only data structure from Section **??** in the protocol in Section **??** provided we assume that there is a bound at which $otsn$ can be corrupted. For example, suppose $otsn$ is bounded by 10, i.e., its possible values are 0..9. Now, if the log says that the value is 5 but the current value is 4, we need to be sure that 5 is the most recent value and it is not the case that the value was increased 5, 6..9, 0..4. That said, in our presentation, we have argued that these two approaches are independent and should not be combined. This is because in Section **??**, we are trying to bound the size of variables used in the protocol. The reason for this is to address the challenge of designing bounded space stabilizing protocol. Now, if we use append-only logs that are by definition unbounded, it violates the spirit of Section **??**.

What happens if the system is dealing with an auditable event during the process recovery? For our purpose, a process that fails and repairs is the equivalent of some process

failing and some other process (with same neighbors as the failed process) that wakes up. In other words, the fact that it was the same process is irrelevant. There are two ways to deal with newly repaired processes: (1) provide guarantees only about processes that are active throughout the restoration process, or (2) delay a process joining the network until its $otsn$ and $ctsn$ values are corrected after repair. In both cases, the correctness of auditable restoration protocol is unchanged due to following reasons:

- If a process fails and recovers in the absence of auditable events, it would obtain $otsn$ and $ctsn$ values from its neighbors. Furthermore existing tree correction algorithms (e.g., [**?**]) permit recovery of processes and incorporate them in the tree. The incorporation of the process in the tree and copying of $otsn$ and $ctsn$ values will preserve all theorems in Section **??**.
- If a process recovers in the presence of auditable events, it will still be eventually incorporated in the tree. This is due to the fact that the tree construction algorithm is independent of auditable events. There are two approaches for dealing with repaired processes. First is to change the correctness requirement to only apply for processes that were working throughout the duration between detection of the auditable event and completion of the restoration via action $AR9$. The second is to delay joining of the repaired process in the network until the leader process is aware of the newly awakened process.
  - In the first way to deal with repaired processes, there is no obligation on behalf of this process. Therefore, correctness of restoration protocol is unaffected.
  - In the second way, this process will have to first get valid $otsn$ and $ctsn$ values, identify the current restoration computation that is going on and notify the leader about its existence. This will allow the leader process to possibly abort the current computation using actions $AR5$ or $AR9$. This will ensure that the properties of the restoration protocol are satisfied.

Protocol AR vs. protocol BAR. The followings are some of the main differences between these two protocols:

- The protocol AR can be enhanced to prevent/manage corruption of $otsn$ and $ctsn$ values. This is due to the fact that these values are unbounded and, hence, we can always detect the 'more recent' values. The protocol BAR can only handle these under some assumptions. For example, in BAR, it is possible that $otsn$ value of all processes is 5. Then, the $otsn$ is increased by actions $BAR1$/$BAR2$ and eventually wraps around to 4. In this case, it is impossible to identify the newest $otsn$ value. In BAR, tolerance to such corruption is provided via stabilization. However, in AR, where $otsn$ is unbounded, we can always distinguish the newer value and, hence, we can handle auditable events at any rate. By contrast, BAR requires one to assume that auditable events occur at a rate that will always allow us to identify the most recent $otsn$.
- Protocol BAR bounds all variables where protocol AR bounds only some.
- Protocol AR is easier to verify since its correctness depends on actions $AR1$ and $AR2$ that cause each process to eventually obtain the maximum $otsn$ value in the system. By contrast, protocol BAR requires an extensive proof [**?**].

APPENDIX B
ASYNCHRONOUS UNISON PROGRAM

In order to better understand our bounding idea described in Section **??**, in this section, we briefly explain the previous work on asynchronous unison in [**?**]. In this program, the constant N denotes the number of processes in the system, and K is any constant greater than N^2. In addition, each process, say p, maintains a public variable $x.p$ whose domain is $[0 \cdots K - 1]$.

Each process p checks the clock values of its neighbors. If the clock value of p is less than that of its neighbor, say q, and the clock values of p and q are close then p checks the clock of its next neighbor, and so on. After p has gone through each of its neighbors successfully, it increments its clock value. If there exists a neighbor, say q, such that $x.p$ and $x.q$ are *far apart* and $x.p$ is higher than $x.q$ then p resets $x.p$ to 0. Thus, the actions of process p in asynchronous unison are as shown in Figure **??**.

$$\forall q : q \in Nbr.p : (x.p \;\; \textit{beh} \;\; x.q]) \;\longrightarrow\; x.p := (x.p + 1)$$

$$\exists q : q \in Nbr.p : ((x.p \;\; \textit{far} \;\; x.q) \;\wedge\; (x.p > x.q)) \longrightarrow x.p := 0$$

where,

$$x.p \;\; \textit{beh} \;\; x.q \;\; \textit{iff} \;\; ((x.q - x.p) \;\; \textit{mod} \;\; K) \leq N$$

$$x.p \;\; \textit{far} \;\; x.q \;\; \textit{iff} \;\; \neg(x.p \; beh \; x.q) \;\wedge\; \neg(x.q \; \textit{beh} \; x.p)$$

Fig. 7: Asynchronous unison protocol.

In [**?**], in it shown that the program represented in Figure **??** is stabilizing in the following sense. Starting from any state, the program is guaranteed to reach a state where the values of neighboring variables are never more than one apart.